УДК 004

В.Ю. Пирогов,
г. Шадринск

Особенности низкоуровневого программирования в 64-битовых операционных системах Windows

В последнее время намечается массовый переход к 64-битовым операционным системам. Статья посвящена особенностям программной модели 64-битовых операционных систем Windows. В качестве инструмента программирования взят кроссплатформенный ассемблер Fasm. Автор останавливается на особенностях 64-битового программирования с использованием данного ассемблера. Во второй части статьи рассматривается вопрос о соглашении вызова процедур в 64-битовых операционных системах Windows. Проводится сравнение с 32-битовыми операционными системами. Подробно рассматривается стандартная структура стека в 64-битовых Windows.

Программирование, ассемблер, Fasm, 64-битовые системы, анализ кода.

V. U. Pirogov,
Shadrinsk

Features of low-level programming in 64-bit Windows operating systems

In recent yearsthere has been a massive migration to 64-bit operating systems. The paper deals with features of the programing model of 64-bit Windowsoperating systems. As a programming tool, the cross-platform assembler Fasmis taken.The author dwells on the peculiarities of 64-bit programming using the assembler Fasm.The second part of the paper is devoted to the issue of calling conventions in 64-bit Windows operating systems.A comparison is made with 32-bit operating systems.The standard stack structure in 64-bit Windows is discussed in detail.

Key words: *programming, assembler, Fasm, 64-bit systems, code analysis.*

Простая низкоуровневая программа

В листинге 1.1 представлена простая программа на языке ассемблера FASM (На момент написания данной главы с авторского сайта (http://flatassembler.net/) можно было скачать версию 1.69.32 для всех платформ, которую я и использую в данной книге. В настоящее время доступна версия 1.71.10.) для операционной системы Windows64. В данной программе имеется только один вызов функции API (подробно об API функциях Windows см. [1,2]). Функция ExitProcess осуществляет корректное завершение текущего процесса (в данном случае просто завершение программы).

Листинг 1.1. Простая программа (Windows64)

```
;описание формата исполняемого модуля
formatPE64 Console
;указать точку входа
entrystart
;секция кода
section '.text' code readable executable
start:
;выделение стека для вызова процедур
;в частности функций API
  sub     rsp,40
;вызов функции API
  mov     rcx,0
  call    [ExitProcess]
;здесь секция данных, если данных нет, то она отсутствует
```

```
section '.data' data readable writeable
  dd       ?
;здесь формируется секция импорта
;она позволяет правильно вызывать функции API
section '.idata' import data readable writeable
  dd 0,0,0,RVA kernel,RVAk_table
  dd 0,0,0,0,0
k_table:
  ExitProcessdq RVA _ExitProcess
  dq 0
  kerneldb 'KERNEL32.DLL',0
  _ExitProcessdw 0
  db 'ExitProcess',0
```

Комментарий к программе из листинга 1.1.

✓ **Трансляция программы**. Для трансляции программы достаточно указать в командной строке запуска FASM.exe имя программы. Вообще программа FASM.exe не имеет параметров, кроме названия программы, да, возможно, имени создаваемого исполняемого модуля. Трансляция, таким образом, может быть запущена просто строкой

FASM.exe prog1.asm [prog.exe]

Конечно, если программа FASM.exe находится не в текущем каталоге, а путь к ней не прописан в окружении (path), то надо указать полный путь. Например, на моем компьютере для трансляции программы я использую такую командную строку.

d:_asm\FASM\windows\FASM.exe prog.asm

Все управление работой транслятора FASM осуществляется на основе директив, которые должны быть расположены в тексте программы.

Если в командной строке отсутствует второй параметр, то компилятор выбирает имя результирующего модуля на основе общих предположений. Например, в Windows, если исходный файл имеет имя prog.asm и описание формата formatPE64, то компилятор автоматически создаст исполняемый модуль с именем prog.exe.

✓ **Основная директива**. Основной директивой, которая управляет работой транслятора FASM, является директива format. В нашем случае мы имеем

format PE64 Console

данная строка говорит транслятору, что требуется создать исполняемый модуль формата PE (PortableExecutable – переносимый исполняемый, формат исполняемых модулей в операционной системе Windows), для 64-битовой системы (окончание 64). Последний элемент строки сообщает транслятору, что должно создаваться консольное приложение, работающее в своей (создаваемой), или чужой консоли.

✓ **Точка входа**. Важный параметр, без которого операционная система не сможет узнать, с какого адреса должна начинать работать программ.

entrystart

Символьная метка start – это и есть точка входа.

✓ **Сегменты (или секции) программы**. Секции - фрагменты программы, которые могут иметь различные свойства и предназначенные для хранения данных, кода, а также служебной информации, необходимой для взаимодействия с внешними модулями, а также другой информации. В нашем случае мы определили всего три секции. Во многих случаях этого будет вполне достаточно. Вот как описывается секция кода.

section '.text' code readable executable

Секция кода. .text – имя секции, code – код программы, readable – для чтения, executable – для исполнения.

В программе мы описали также секцию данных. Для секции данных мы указали также флаг writeable, другими словами, мы разрешаем делать запись в эту секцию. Важнейшей для нас будет секция импорта. Она необходима нам, чтобы подключаться к системным динамическим библиотекам. Секция импорта имеет строго заданную структура, которая содержит имя загружаемой динамической библиотеки (в нашем случае KERNEL32.DLL) и имя API функции (в нашем случае ExitProcess). Предположим, что нам надо использовать еще одну функцию API с именем app, которая располагается в динамической библиотеке LIBR2.DLL. Тогда наша секция импорта была бы такой

```
section '.idata' import data readable writeable
dd 0,0,0,RVA kernel, RVA k_table
dd 0,0,0,RVA libr2,RVA l_table
dd 0,0,0,0,0
k_table:
ExitProcessdq RVA _ExitProcess
dq 0
l_table:
appdq RVA _app
dq 0
kerneldb 'KERNEL32.DLL',0
libr2db 'LIBR2.DLL',0
_ExitProcessdw 0
db 'ExitProcess',0
_app dw 0
db 'app',0
```

Отметим, только, используемое обозначение RVA – RelativeVirtualAddress (относительный виртуальный адрес). Это адрес в виртуальном адресном пространстве относительно адреса загрузки исполняемого модуля. Подробное описание структуры исполняемого модуля для Windows можно найти в книге [3], см. также [7,8].

✓ Вызов функции API осуществляется обычной процессорной инструкцией, которая на ассемблере обозначается словом call. Далее, в статье, мы подробно разберем, как в Windows64 вызываются процедуры и в частности API функции.

✓ При написании программы можно использовать не только функции API, но и другие процедуры, которые динамически (во время выполнения программы) или статически (во время трансляции) будут подключаться к исполняемому модулю. Вызов таких процедур принципиально ничем не отличается от вызовов функций API ([2]).

Вызов процедур в Windows64

Функции API нельзя в полной мере назвать системными функциями, так как обычно под системными функциями понимаю функции, предоставляемые самой системой. API же Windows являются некоторой прослойкой, интерфейсом, между прикладной программой и операционной системой. Количество их в Windows огромно и охватывает практически все вопросы, так или иначе связанные с нуждами прикладной, да и системной программы (Полный перечень API-функций для Windowsможно найти на сайте MSDN - https://msdn.microsoft.com/en-us/library/cc433218).

Функции API в Windows реализованы через набор системных динамических библиотек, большинство из которых расположены в %systemroot%\System32 и подключаются посредством динамического связывания во время запуска

исполняемого модуля. В 32-битовой операционной системе Windows реализованы два альтернативных способа вызова API функций. В обоих случаях параметры передаются через стек. Общая схема вызова функции f_api(parn,parn-1...par2,par1) такая (см. [4, 5]):

```
push par1
push par2
...
push parn-1
pushparn
callf_api
```

Как видим, здесь действует простое правило: "слева направо, снизу-вверх".

При вызове функции командой call, как известно, в стек кладется адрес возврата. В отсутствие параметров при возврате из функции (команда ret, правильнее было бы сказать retn (возврат в пределах одного сегмента)) адрес извлекается из стека, т.е. положение вершины стека восстанавливается. Однако, если в стек предварительно положили параметры (n - количество параметров), то после вызова функции указатель стека остается смещенным на величину 4*n. При многократном вызове функций с параметром стек может быть исчерпан, что приведет к краху программы. Для возвращения стека в исходное состояние используется два механизма.

1. Нотация языка Паскаль (см., например, [6]). Стек восстанавливает вызываемая функция, путем использования команды rett, где t=n*4. Эта нотация используется в большинстве функций API Win32.

2. Нотация языка C (или нотация cdecl) (см. [3]). Стек восстанавливает вызывающая сторона. После вызова команды call, для восстановления стека применяется команда addesp,t, где t=4*n. Подобный механизм используется для функций с переменным числом параметров, например, wsprintf (функция осуществляет копирование форматной строки в буфер, подставляя туда значения параметров). При этом порядок отправки параметров в стек иной. Здесь работает правило "справа налево, снизу-вверх".

Еще один вопрос, который важен при использовании функции API - какие регистры процессора сохраняются после вызова функции и каким образом данные возвращаются из функции. Для 32-битовой Windows принято сохранение регистров ebx, esi, edi, ebp. Возврат данных из функции осуществляется посредством регистра eax, пары edx:eax или регистра сопроцессора st0 (в случае типа float. В 32-х битовых операционных системах не регламентировалось какое-либо конкретное соглашение о вызовах. По этой причине появилось несколько разных соглашений. Неко-торые соглашения были присущи только одному конкретному компилятору (см. [2, 5]). Для того, чтобы пользоваться какой-либо библиотекой, необходимо было знать, какую нотацию следует использовать при вызове хранящихся в библиотеке процедур. Иногда было не просто использовать такие библиотеки в программе на языке высокого уровня).

Что касается 64-битовых Windows, то в них принята всего одна конвенция вызова процедур, в основе, которой лежит передача параметров через регистры. Несомненно, такой подход упрощает проблемы программистов, в частности при использовании в своей программе сторонних библиотек. Рассмотрим основные положения данного соглашения.

✓ Первые четыре параметра передаются в функцию через регистры: rcx, rdx, r8, r9. Остальные параметры (если они есть) передаются через стек.

✓ Перед вызовом функции резервируется область в стеке (теневая область), на случай, если вызываемая функция "захочет" временно сохранить параметры в

стеке. Те же параметры, которые передаются через стек, помещаются в старших адресах выделенной области.

✓ При передаче параметров, размер которых меньше 64 бит, передаются как 64-битовые параметры. При этом следует обнулить старшие биты. Параметры, большие 64-бит передаются по ссылке.

✓ Данные возвращаются через регистр rax. Если возвращаемое значение имеет размер больший 64 бит, то данное передается через область памяти, адрес которой передается в первом параметре.

✓ Все регистры при вызове функций сохраняются за исключением rax, rcx, rdx, r8, r9, r10, r11, сохранность которых не гарантируется.

✓ Стек восстанавливается вызывающей стороной, обычно это команда addrsp,n , где n – размер, ранее выделенной области стека.

✓ Стек должен быть выровнен на величину кратную 16 (правильнее сказать, что вершина (указатель) стека должна иметь адрес кратный 16). Следует учитывать, что при вызове в стеке сохраняется также и адрес возврата, занимающий 8 байт. Следовательно, перед вызовом выделяется область памяти равная 16*n+8 (см. ниже), где n некоторое целое число.

✓ Соглашением также предусмотрено, что для передачи параметров могут использоваться четыре 128-ми битовых регистров xmm0, xmm1, xmm2, xmm3. Они должны использоваться для передачи чисел, разрядность которых больше чем 64-бита. Однако общее количество регистров, используемых для передачи параметров все равно должно быть 4. Таким образом, если первые четыре параметра передаются регистрами общего назначения, то регистры xmm0-xmm3 вообще не используются. Если, например, второй параметр имеет разрядность, большую, чем 64, то для его передачи должен использовать регистр xmm1, но тогда регистр rdx для передачи параметров уже использоваться не будет.

Рассмотрим в общих чертах схему вызова функции API с пятью параметрами.

```
...
subrsp,40 ; резервируем стек
movqwordptr [rsp+32],par5
mov r9,par4
mov r8,par3,
mov rdx,par2
movrcx,par1
callf_api64
...
addrsp,40 ;восстанавливаем стек
...
```

Обратите внимание, что в нашем случае стек с учетом адреса возврата при вызове функции оказывается выровненным на величину кратную 16 (48 байт). С учетом этого выравнивания вызов функции, содержащей только 4 параметра, будет выглядеть так

```
subrsp,40 ; резервируем стек
movr9,par4
mov r8,par3,
mov rdx,par2
movrcx,par1
call f_api64
...
addrsp,40 ;восстанавливаем стек
```

На *рисунке 1* представлена схема стека при вызове функции в 64-битовой Windows. Из рисунка видно, что по сути хотя четыре параметра и передаются

посредством регистров, для них все равно отводится место в стеке (теневая область). Если количество параметров больше чем 4, то они помещаются в стек уже обычным способом. Не следует забывать, что стек в любом случае должен быть выровнен по значению равному 16 (речь идет о значении, которое лежит в регистре rsp). Это значит, что даже если параметров нет, то в стеке должна быть выделена дополнительно область в 8 байтов (довольно часто в начале процедуры стоит команда pushrbp, которая и дает дополнитель-ный сдвиг стека на 8 байтов).

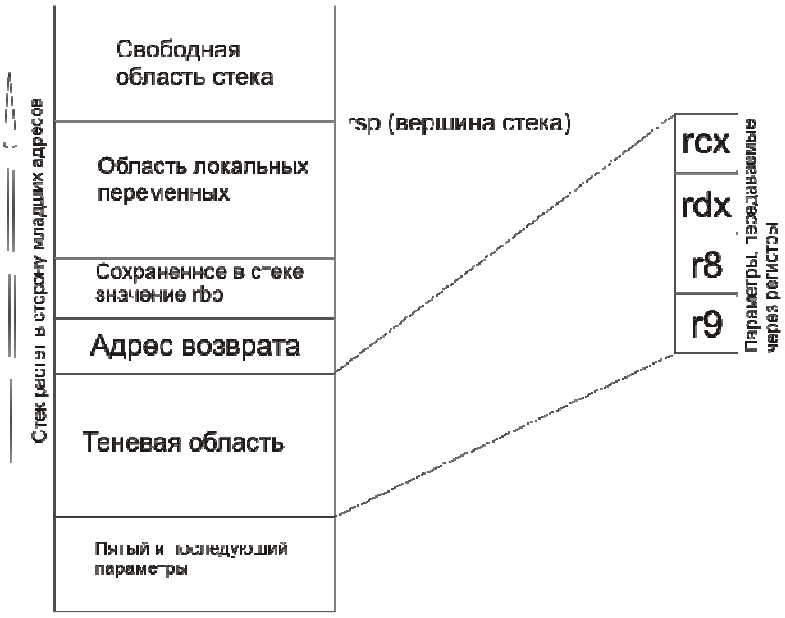

Рисунок 1. Структура стека при вызове функции в Windows64

Зарезервировать и одновременно выровнять стек можно один раз, в начале кода, сразу зарезервировав достаточное количество байтов, например, командой subrsp,88. При этом нет нужды возвращать стек в исходное состояние после вызова каждой функции. Если код заканчивается вызовом функции ExitProcess, то и вообще команда addrsp,88 не нужна, так как функция не возвращает управление текущему процессу, а закрывает его. Если в программе имеются процедуры, в которых есть вызов API-функций или других процедур, то в начале процедуры также должна стоять команда резервирования стека subrsp,n, а в конце команда addrsp,n .

Заметим, что использование нового соглашения о вызовах не приводит к экономии стека, а скорее наоборот. Каждый вызов процедуры расходует определенное количество стека, при этом зарезервированная область стека часто никак не используется. Не забудем также, что локальные переменные также хранятся в стеке. С другой стороны, использование регистров для передачи параметров может несколько повысить скорость вызова процедуры и в целом производительность программы.

Следует также иметь в виду, что стек используется также для хранения локальных переменных. Если обратиться к рисунку 1, то для локальных переменных будет использована область, названная остатком стека. Также как и теневая область, область для хранения локальных переменных выделяется командой subrsp,n. Если нужна и теневая область, и область хранения локальных переменных то, разумеется, общая область выделяется одной командой.

Конечно, если процедура пишется на языке ассемблера и используется только в самой программе, то можно придерживаться любой конвенции вызова, в том числе и придуманной лично автором программы. Однако не стоит этим увлекаться, поскольку написанные процедуры могут понадобиться другой программе и тогда придется их адаптировать. Заметим, что выделяемая в стеке

резервная область может и не использоваться в вызванной процедуре для хранения параметров (некоторые компиляторы с языков высокого уровня так и поступают).

ЛИТЕРАТУРА

1. Зубков, С.В. Assembler для DOS, Windows и Unix [Текст] / С.В. Зубков. – М. : БХВ, 2000.

2. Пирогов, В.Ю. Ассемблер для Windows [Текст] / В.Ю. Пирогов. – Издание 4-е. – СПб. : БХВ, 2007.

3. Пирогов, В.Ю. Ассемблер и дизассемблирование [Текст] / В.Ю. Пирогов. – СПб. : БХВ, 2006.

4. Пирогов, В.Ю. Операционные системы на базе набора команд x86–64 в контексте низкоуровневого программирования [Текст] /В.Ю. Пирогов // Прикладная информатика. – № 6 (60). – 2015.

5. Calling Convention [Electronic resource] // MSDN. – Access mode: http://msdn.microsoft.com/en-us/library/9b372w95.aspx. – 13.01.2015.

6. Lazarus documentation [Electronic resource] – Access mode: http://wiki.lazarus.freepascal.org/Lazarus_Documentation. – 13.01.2015.

7. Pietrek, Matt. A Crash Course on the Depths of Win32™ Structured Exception Handling [Electronic resource] / Matt Pietrek. // Microsoft system journal. – 1997. – Access mode: http://www.microsoft.com/msj/0197/exception/exception.aspx. – 13.01.2015.

8. Running 32-bit Applications [Electronic resource] // MSDN. – Access mode: http://msdn.microsoft.com/en-us/library/aa384249(VS.85).aspx. – 13.01.2015.